**UNIT IV**

Functions are a construct to structure programs. They are known in most programming languages, sometimes also called subroutines or procedures. Functions are used to utilize code in more than one place in a program.

A function in Python is defined by a “def ” statement.

**Syntax:**

def function-name(Parameter list):

statements

The parameter list consists of none or more parameters. Parameters are called arguments, if the function is called. The function body consists of indented statements. The function body gets executed every time the function is called.

Parameter can be mandatory or optional. The optional parameters (zero or more) must follow the mandatory parameters.

* Keyword def marks the start of function header.
* A function name is to uniquely identify it.
* Function naming follows the same [rules of writing identifiers in Python](https://www.programiz.com/python-programming/keywords-identifier#rules).
* Parameters (arguments) are which we pass values to a function.
* A colon (:) to mark the end of function header.
* Optional documentation string (docstring) to describe what the function does.
* One or more valid python statements that make up the function body.
* Statements must have same indentation level (usually 4 spaces).
* An optional return statement to return a value from the function.

You use functions in programming to bundle a set of instructions that you want to use repeatedly or that, because of their complexity, are better self-contained in a sub-program and called when needed. That means that a function is a piece of code written to carry out a specified task. To carry out that specific task, the function might or might not need multiple inputs. When the task is carried out, the function can or cannot return one or more values.

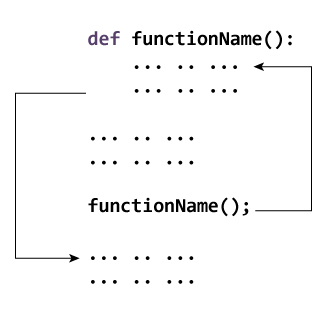
**Types of functions:**

There are three types of functions in Python:

1. **Built-in functions:** such as help() to ask for help, min() to get the minimum value, print() to print an object to the terminal,…
2. **User-Defined Functions (UDFs):** which are functions that users create to help them out;
3. **Anonymous functions:** which are also called lambda functions because they are not declared with the standard def keyword.

**User-Defined Functions (UDFs):**

**How Function works in Python?**

* 

**Advantages of Functions:**

The advantages of using functions are:

* Reducing duplication of code
* Decomposing complex problems into simpler pieces
* Improving clarity of the code
* Reuse of code
* Information hiding

Functions in Python are first-class citizens. It means that functions have equal status with other objects in Python. Functions can be assigned to variables, stored in collections, or passed as arguments. This brings additional flexibility to the language.

**Calling a Function**

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt.

**Example to call printme() function:**

# Function definition is here

def printme( str ):

"This prints a passed string into this function"

print (str)

return

# Now you can call printme function

printme("I'm first call to user defined function!")

printme("Again second call to the same function")

When the above code is executed, the output is as follows:

I'm first call to user defined function!

Again second call to the same function

**Pass by reference vs value**

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function.

**Example:**

# Function definition is here

def changeme(mylist):

"This changes a passed list into this function"

mylist.append([1,2,3,4]);

print ("Values inside the function: ", mylist)

return

# Now you can call changeme function

mylist = [10,20,30];

changeme( mylist );

print ("Values outside the function: ", mylist)

Here, we are maintaining reference of the passed object and appending values in the same object. So, the output is:

Values inside the function: [10, 20, 30, [1, 2, 3, 4]]

Values outside the function: [10, 20, 30, [1, 2, 3, 4]]

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.

# Function definition is here

def changeme(mylist):

"This changes a passed list into this function"

mylist = [1,2,3,4]; # This would assign new reference in mylist

print ("Values inside the function: ", mylist)

return

# Now you can call changeme function

mylist = [10,20,30];

changeme( mylist );

print ("Values outside the function: ", mylist)

The parameter mylist is local to the function changeme. Changing mylist within the function does not affect mylist. The function accomplishes nothing and finally the output is:

Values inside the function: [1, 2, 3, 4]

Values outside the function: [10, 20, 30]

**Function Arguments**

You can call a function by using the following types of formal arguments:

* Required arguments
* Keyword arguments
* Default arguments
* Variable-length arguments

**Required arguments**

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call the function printme(), you definitely need to pass one argument, otherwise it gives a syntax error as follows −

# Function definition is here

def printme( str ):

"This prints a passed string into this function"

print (str)

return;

# Now you can call printme function

printme()

When the above code is executed, the output is as follows:

Traceback (most recent call last):

File "test.py", line 11, in <module>

printme();

TypeError: printme() takes exactly 1 argument (0 given)

**Keyword arguments**

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the printme() function as:

# Function definition is here

def printme( str ):

"This prints a passed string into this function"

print (str)

return

# Now you can call printme function

printme( str = "My string")

When the above code is executed, the output is as follows:

My string

**Another example:**

Note that the order of parameters does not matter.

# Function definition is here

def printinfo( name, age ):

"This prints a passed info into this function"

print ("Name: ", name)

print ("Age ", age)

return

# Now you can call printinfo function

printinfo( age=50, name="miki" )

When the above code is executed, the output is:

Name: miki

Age 50

**Default arguments**

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.

**Example:**

# Function definition is here

def printinfo( name, age = 35 ):

"This prints a passed info into this function"

print ("Name: ", name)

print ("Age ", age)

return

# Now you can call printinfo function

printinfo( age=50, name="miki" )

printinfo( name="miki" )

When the above code is executed, the output is as follows:

Name: miki

Age 50

Name: miki

Age 35

**Variable-length arguments**

You may need to process a function for more arguments than you specified while defining the function. These arguments are called variable-length arguments and are not named in the function definition, unlike required and default arguments.

**Syntax:**

def functionname([formal\_args,] \*var\_args\_tuple ):

"function\_docstring"

function\_suite

return [expression]

An asterisk (\*) is placed before the variable name that holds the values of all non-keyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call.

**Example:**

# Function definition is here

def printinfo( arg1, \*vartuple ):

"This prints a variable passed arguments"

print ("Output is: ")

print (arg1)

for var in vartuple:

print var

return;

# Now you can call printinfo function

printinfo( 10 )

printinfo( 70, 60, 50 )

When the above code is executed, the output is:

10

70

60

50

**The Anonymous Functions**

These functions are called anonymous because they are not declared in the standard manner by using the def keyword. You can use the lambda keyword to create small anonymous functions.

* Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
* An anonymous function cannot be a direct call to print because lambda requires an expression.
* Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
* Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

**Syntax:**

The syntax of lambda functions contains only a single statement, which is as follows:

lambda [arg1 [,arg2,.....argn]]:expression

**Example:**

# Function definition is here

sum = lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function

print "Value of total : ", sum( 10, 20 )

print "Value of total : ", sum( 20, 20 )

When the above code is executed, the output is as follows:

Value of total : 30

Value of total : 40

**The return Statement**

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

All the above examples are not returning any value. You can return a value from a function as follows –

# Function definition is here

def sum( arg1, arg2 ):

# Add both the parameters and return them."

total = arg1 + arg2

print ("Inside the function : ", total)

return total

# Now you can call sum function

total = sum( 10, 20 );

print ("Outside the function : ", total)

When the above code is executed, the output is:

Inside the function : 30

Outside the function : 30

**Scope of Variables**

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier.

There are two basic scopes of variables in Python:

* Global variables
* Local variables

**Global vs. Local variables:**

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope.

**Example:**

total = 0; # This is global variable.

# Function definition is here

def sum( arg1, arg2 ):

# Add both the parameters and return them."

total = arg1 + arg2; # Here total is local variable.

print ("Inside the function local total : ", total)

return total

# Now you can call sum function

sum( 10, 20 )

print ("Outside the function global total : ", total)

When the above code is executed, the output is:

Inside the function local total : 30

Outside the function global total : 0

**Scope and Lifetime of variables:**

Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function is not visible from outside. Hence, they have a local scope.

Lifetime of a variable is the period throughout which the variable exits in the memory. The lifetime of variables inside a function is as long as the function executes.

They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

**Example to illustrate the scope of a variable inside a function:**

def my\_func():

x = 10

print("Value inside function:",x)

x = 20

my\_func()

print("Value outside function:",x)

Output:

Value inside function: 10

Value outside function: 20

Here, we can see that the value of x is 20 initially. Even though the function my\_func() changed the value of x to 10, it did not effect the value outside the function.

This is because the variable x inside the function is different (local to the function) from the one outside. Although they have same names, they are two different variables with different scope.

On the other hand, variables outside of the function are visible from inside. They have a global scope.

We can read these values from inside the function but cannot change (write) them. In order to modify the value of variables outside the function, they must be declared as global variables using the keyword global.

**Docstring (Documentation Strings)**

Python documentation strings (or docstrings) provide a convenient way of associating documentation with Python modules, functions, classes, and methods. It’s specified in source code that is used, like a comment, to document a specific segment of code. Unlike conventional source code comments, the docstring should describe what the function does, not how.

An object's docsting is defined by including a string constant as the first statement in the object's definition. It's specified in source code that is used, like a comment, to document a specific segment of code.

Unlike conventional source code comments the docstring should describe what the function does, not how.

* All functions should have a docstring.
* This allows the program to inspect these comments at run time, for instance as an interactive help system, or as metadata.
* Docstrings can be accessed by the \_\_doc\_\_ attribute on objects.
* The doc string line should begin with a capital letter and end with a period.
* The first line should be a short description.
* Don't write the name of the object.
* If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description.
* The following lines should be one or more paragraphs describing the object’s calling conventions, its side effects, etc.

def my\_function():

"""Do nothing, but document it.

No, really, it doesn't do anything. """

pass

This would look like when we print it:

>>> print my\_function.\_\_doc\_\_

Do nothing, but document it. No, really, it doesn't do anything.

**Declaring Docstrings:**

The docstrings are declared using “””triple double quotes””” just below the class, method or function declaration. All functions should have a docstring.

The following Python file shows the declaration of docstrings within a python source file:

""" Assuming this is file mymodule.py, then this string, being the first statement in the file, will become the "mymodule" module's docstring when the file is imported."""

class MyClass(object):

"""The class's docstring"""

def my\_method(self):

"""The method's docstring"""

def my\_function():

"""The function's docstring"""

**Accessing Docstrings:**

The docstrings can be accessed using the \_\_doc\_\_ method of the object or using the help function.

The below example demonstrates how to declare and access a docstring.

|  |
| --- |
| **def** my\_function():       """Demonstrate docstrings and does nothing really."""    **return** None    print "Using \_\_doc\_\_:"  print my\_function.\_\_doc\_\_    print "Using help:"  help(my\_function) |

Output:

Using \_\_doc\_\_:

Demonstrate docstrings and does nothing really.

Using help:

Help on function my\_function in module \_\_main\_\_:

my\_function()

Demonstrate docstrings and does nothing really.

>>> import mymodule

>>> help(mymodule)

Assuming this is file mymodule.py then this string, being the first statement in the file will become the mymodule modules docstring when the file is imported.

>>> help(mymodule.MyClass)

The class's docstring

>>> help(mymodule.MyClass.my\_method)

The method's docstring

>>> help(mymodule.my\_function)

The function's docstring

**One-line Docstrings:**

As the name suggests, one line docstrings fit in one line. They are used in obvious cases. The closing quotes are on the same line as the opening quotes. This looks better for one-liners.

**For example:**

|  |
| --- |
| def power(a, b):      """Returns arg1 raised to power arg2."""        return a**\*\***b    print power.\_\_doc\_\_ |

Output:

Returns arg1 raised to power arg2.

**Multi-line Docstrings:**

Multi-line docstrings consist of a summary line just like a one-line docstring, followed by a blank line, followed by a more elaborate description. The summary line may be on the same line as the opening quotes or on the next line.

The example below shows a multi-line docstring.

|  |
| --- |
| def my\_function(arg1):      """      Summary line.        Extended description of function.        Parameters:      arg1 (int): Description of arg1        Returns:      int: Description of return value        """        return arg1    print my\_function.\_\_doc\_\_ |

Output:

Summary line.

Extended description of function.

Parameters:

arg1 (int): Description of arg1

Returns:

int: Description of return value

**Indentation in Docstrings:**

The entire docstring is indented the same as the quotes at its first line. Docstring processing tools will strip a uniform amount of indentation from the second and further lines of the docstring, equal to the minimum indentation of all non-blank lines after the first line. Any indentation in the first line of the docstring (i.e., up to the first newline) is insignificant and removed. Relative indentation of later lines in the docstring is retained.

**Docstrings in Classes:**

Let us take an example to show how to write docstrings for a class and its methods. helpis used to access the docstring.

|  |
| --- |
| class ComplexNumber:      """      This is a class for mathematical operations on complex numbers.        Attributes:          real (int): The real part of complex number.          imag (int): The imaginary part of complex number.      """        def \_\_init\_\_(self, real, imag):          """          The constructor for ComplexNumber class.            Parameters:             real (int): The real part of complex number.             imag (int): The imaginary part of complex number.          """        def add(self, num):          """          The function to add two Complex Numbers.            Parameters:              num (ComplexNumber): The complex number to be added.            Returns:              ComplexNumber: A complex number which contains the sum.          """            re **=** self.real **+** num.real          im **=** self.imag **+** num.imag            return ComplexNumber(re, im)    help(ComplexNumber)  # to access Class docstring  help(ComplexNumber.add)  # to access method's docstring |

Output:

Help on class ComplexNumber in module \_\_main\_\_:

class ComplexNumber

| This is a class for mathematical operations on complex numbers.

|

| Attributes:

| real (int): The real part of complex number.

| imag (int): The imaginary part of complex number.

|

| Methods defined here:

|

| \_\_init\_\_(self, real, imag)

| The constructor for ComplexNumber class.

|

| Parameters:

| real (int): The real part of complex number.

| imag (int): The imaginary part of complex number.

|

| add(self, num)

| The function to add two Complex Numbers.

|

| Parameters:

| num (ComplexNumber): The complex number to be added.

|

| Returns:

| ComplexNumber: A complex number which contains the sum.

Help on method add in module \_\_main\_\_:

add(self, num) unbound \_\_main\_\_.ComplexNumber method

The function to add two Complex Numbers.

Parameters:

num (ComplexNumber): The complex number to be added.

Returns:

ComplexNumber: A complex number which contains the sum.

**Recursive functions:**

Recursion is the process of defining something in terms of itself.

A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

**Python Recursive Function:**

We know that in Python, a [function](https://www.programiz.com/python-programming/function) can call other functions. It is even possible for the function to call itself. These type of construct are termed as recursive functions.

Following is an example of recursive function to find the factorial of an integer.

Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is 1\*2\*3\*4\*5\*6 = 720.

def calc\_factorial(x):

"""This is a recursive function to find the factorial of an integer"""

if x == 1:

return 1

else:

return (x \* calc\_factorial(x-1))

num = 4

print("The factorial of", num, "is", calc\_factorial(num))

In the above example, calc\_factorial() is a recursive functions as it calls itself.

When we call this function with a positive integer, it will recursively call itself by decreasing the number.

Each function calls multiples the number with the factorial of number 1 until the number is equal to one.

This recursive call can be explained in the following steps.

calc\_factorial(4) # 1st call with 4

4 \* calc\_factorial(3) # 2nd call with 3

4 \* 3 \* calc\_factorial(2) # 3rd call with 2

4 \* 3 \* 2 \* calc\_factorial(1) # 4th call with 1

4 \* 3 \* 2 \* 1 # return from 4th call as number=1

4 \* 3 \* 2 # return from 3rd call

4 \* 6 # return from 2nd call

24 # return from 1st call

Our recursion ends when the number reduces to 1. This is called the base condition.

Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

**Advantages of Recursion:**

* Recursive functions make the code look clean and elegant.
* A complex task can be broken down into simpler sub-problems using recursion.
* Sequence generation is easier with recursion than using some nested iteration.

**Disadvantages of Recursion**

* Sometimes the logic behind recursion is hard to follow through.
* Recursive calls are expensive (inefficient) as they take up a lot of memory and time.

Recursive functions are hard to debug.

The highest common factor (H.C.F) or greatest common divisor (G.C.D) of two numbers is the largest positive integer that perfectly divides the two given numbers.

**For example, the H.C.F of 12 and 14 is 2.**

# define a function

def computeHCF(x, y):

# choose the smaller number

if x > y:

smaller = y

else:

smaller = x

for i in range(1, smaller+1):

if((x % i == 0) and (y % i == 0)):

hcf = i

return hcf

# take input from the user

num1 = int(input("Enter first number: "))

num2 = int(input("Enter second number: "))

print("The H.C.F. of", num1,"and", num2,"is", computeHCF(num1, num2))

Here, two integers stored in variables num1 and num2 are passed to a function which returns the H.C.F.

In the function, we first determine the smaller of the two number since the H.C.F can only be less than or equal to the smallest number. We then use for loop to go from 1 to that number.

In each iteration, we check if our number perfectly divides both the input numbers. If so, we store the number as H.C.F. At the completion of the loop we end up with the largest number that perfectly divides both the numbers.

**Modules:**

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.

Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.

**Example:**

The Python code for a module named aname normally resides in a file named aname.py. Here's an example of a simple module, support.py

def print\_func( par ):

print ("Hello : ", par)

return

**The import Statement:**

You can use any Python source file as a module by executing an import statement in some other Python source file.

**The import has the following syntax:**

import module1[, module2[,... moduleN]

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module. For example, to import the module support.py, you need to put the following command at the top of the script –

# Import module support

import support

# Now you can call defined function that module as follows

support.print\_func("Zara")

When the above code is executed, the output is:

Hello : Zara

A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening over and over again if multiple imports occur.

**The from...import Statement**

Python's from statement lets you import specific attributes from a module into the current namespace.

**The from...import has the following syntax:**

from modname import name1[, name2[, ... nameN]]

For example, to import the function fibonacci from the module fib:

from fib import fibonacci

This statement does not import the entire module fib into the current namespace; it just introduces the item fibonacci from the module fib into the global symbol table of the importing module.

**The from...import \* Statement**

It is also possible to import all names from a module into the current namespace by using the following import statement:

from modname import \*

This provides an easy way to import all the items from a module into the current namespace; however, this statement should be used sparingly.

**Locating Modules**

When you import a module, the Python interpreter searches for the module in the following sequences:

* The current directory.
* If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
* If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python/.

The module search path is stored in the system module sys as the **sys.path** variable. The sys.path variable contains the current directory, PYTHONPATH, and the installation-dependent default.

**The PYTHONPATH Variable**

The PYTHONPATH is an environment variable, consisting of a list of directories. The syntax of PYTHONPATH is the same as that of the shell variable PATH.

Here is a typical PYTHONPATH from a Windows system:

set PYTHONPATH = c:\python20\lib;

And here is a typical PYTHONPATH from a UNIX system:

set PYTHONPATH = /usr/local/lib/python

**Namespaces and Scoping**

Variables are names (identifiers) that map to objects. A namespace is a dictionary of variable names (keys) and their corresponding objects (values).

A Python statement can access variables in a local namespace and in the global namespace. If a local and a global variable have the same name, the local variable shadows the global variable.

Each function has its own local namespace. Class methods follow the same scoping rule as ordinary functions.

Python makes educated guesses on whether variables are local or global. It assumes that any variable assigned a value in a function is local.

Therefore, in order to assign a value to a global variable within a function, you must first use the global statement.

The statement global VarName tells Python that VarName is a global variable. Python stops searching the local namespace for the variable.

For example, we define a variable Money in the global namespace. Within the function Money, we assign Money a value, therefore Python assumes Money as a local variable. However, we accessed the value of the local variable Money before setting it, so an UnboundLocalError is the result. Uncommenting the global statement fixes the problem.

Money = 2000

def AddMoney():

global Money

Money = Money + 1

print (Money)

AddMoney()

print (Money)

**The dir( ) Function**

The dir() built-in function returns a sorted list of strings containing the names defined by a module.

The list contains the names of all the modules, variables and functions that are defined in a module.

Example:

# Import built-in module math

import math

content = dir(math)

print content

When the above code is executed, it produces the following result −

['\_\_doc\_\_', '\_\_file\_\_', '\_\_name\_\_', 'acos', 'asin', 'atan',

'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',

'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',

'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',

'sqrt', 'tan', 'tanh']

Here, the special string variable \_\_name\_\_ is the module's name, and \_\_file\_\_ is the filename from which the module was loaded.

**The globals() and locals() Functions:**

The globals() and locals() functions can be used to return the names in the global and local namespaces depending on the location from where they are called.

If locals() is called from within a function, it will return all the names that can be accessed locally from that function.

If globals() is called from within a function, it will return all the names that can be accessed globally from that function.

The return type of both these functions is dictionary. Therefore, names can be extracted using the keys() function.

**The reload() Function:**

When the module is imported into a script, the code in the top-level portion of a module is executed only once.

Therefore, if you want to re-execute the top-level code in a module, you can use the reload() function. The reload() function imports a previously imported module again.

The **syntax** of the reload() function:

reload(module\_name)

Here, module\_name is the name of the module you want to reload and not the string containing the module name.

For example, to reload hello module, do the following −

reload(hello)

**Packages in Python**

A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and sub-packages and sub-packages, and so on.

Consider a file Pots.py available in Phone directory. This file has following line of source code −

def Pots():

print "I'm Pots Phone"

Similar way, we have another two files having different functions with the same name as above:

* Phone/Isdn.py file having function Isdn()
* Phone/G3.py file having function G3()

Now, create one more file \_\_init\_\_.py in Phone directory:

* Phone/\_\_init\_\_.py

To make all of your functions available when you've imported Phone, you need to put explicit import statements in \_\_init\_\_.py as follows:

from Pots import Pots

from Isdn import Isdn

from G3 import G3

After you add these lines to \_\_init\_\_.py, you have all of these classes available when you import the Phone package.

# Now import your Phone Package.

import Phone

Phone.Pots()

Phone.Isdn()

Phone.G3()

When the above code is executed, the output is:

I'm Pots Phone

I'm 3G Phone

I'm ISDN Phone

In the above example, we have taken example of a single functions in each file, but you can keep multiple functions in your files. You can also define different Python classes in those files and then you can create your packages out of those classes.

The Python Package Index (PyPI) is a repository of software for the Python programming language. PyPI helps you find and install software developed and shared by the Python community. Learn about installing packages. Package authors use PyPI to distribute their software.

**String functions:**

Python has several built-in functions associated with the [string data type](https://www.digitalocean.com/community/tutorials/an-introduction-to-working-with-strings-in-python-3). These functions let us easily modify and manipulate strings. We can think of functions as being actions that we perform on elements of our code. Built-in functions are those that are defined in the Python programming language and are readily available for us to use.

**Making Strings Upper and Lower Case:**

The functions str.upper() and str.lower() will return a string with all the letters of an original string converted to upper- or lower-case letters. Because strings are immutable data types, the returned string will be a new string. Any characters in the string that are not letters will not be changed.

Let’s convert the string Sammy Shark to be all upper case:

ss = "Sammy Shark"

print(ss.upper())

Output:

SAMMY SHARK

Now, let’s convert the string to be all lower case:

print(ss.lower())

Output:

sammy shark

The str.upper() and str.lower() functions make it easier to evaluate and compare strings by making case consistent throughout. That way if a user writes their name all lower case, we can still determine whether their name is in our database by checking it against an all upper-case version, for example.

**Boolean Methods:**

Python has some string methods that will evaluate to a [Boolean value](https://www.digitalocean.com/community/tutorials/understanding-boolean-logic-in-python-3). These methods are useful when we are creating forms for users to fill in, for example. If we are asking for a postcode we will only want to accept a numeric string, but when we are asking for a name, we will only want to accept an alphabetic string.

There are a number of string methods that will return Boolean values:

|  |  |
| --- | --- |
| Method | **True** if |
| str.isalnum() | String consists of only alphanumeric characters (no symbols) |
| str.isalpha() | String consists of only alphabetic characters (no symbols) |
| str.islower() | String’s alphabetic characters are all lower case |
| str.isnumeric() | String consists of only numeric characters |
| str.isspace() | String consists of only whitespace characters |
| str.istitle() | String is in title case |
| str.isupper() | String’s alphabetic characters are all upper case |

Let’s look at a couple of these in action:

number = "5"

letters = "abcdef"

print(number.isnumeric())

print(letters.isnumeric())

Output:

True

False

Using the str.isnumeric() method on the string 5 returns a value of True, while using the same method on the string abcdef returns a value of False.

Similarly, we can query whether a string’s alphabetic characters are in title case, upper case, or lower case. Let’s create a few strings:

movie = "2001: A SAMMY ODYSSEY"

book = "A Thousand Splendid Sharks"

poem = "sammy lived in a pretty how town"

Now let’s try the Boolean methods that check for case:

print(movie.islower())

print(movie.isupper())

print(book.istitle())

print(book.isupper())

print(poem.istitle())

print(poem.islower())

Now we can run these small programs and see the output:

Output of movie string

False

True

Output of book string

True

False

Output of poem string

False

True

Checking whether characters are lower case, upper case, or title case, can help us to sort our data appropriately, as well as provide us with the opportunity to standardize data we collect by checking and then modifying strings as needed.

Boolean string methods are useful when we want to check whether something a user enters fits within given parameters.

**Determining String Length:**

The string method len() returns the number of characters in a string. This method is useful for when you need to enforce minimum or maximum password lengths, for example, or to truncate larger strings to be within certain limits for use as abbreviations.

To demonstrate this method, we’ll find the length of a sentence-long string:

open\_source = "Sammy contributes to open source."

print(len(open\_source))

Output:

33

We set the variable open\_source equal to the string "Sammy contributes to open source." and then we passed that variable to the len() method with len(open\_source). We then passed the method into the print() method so that we could see the output on the screen from our program.

Keep in mind that any character bound by single or double quotation marks — including letters, numbers, whitespace characters, and symbols — will be counted by the len() method.

**join(), split(), and replace() Methods:**

The str.join(), str.split(), and str.replace() methods are a few additional ways to manipulate strings in Python.

The str.join() method will concatenate two strings, but in a way that passes one string through another.

Let’s create a string:

balloon = "Sammy has a balloon."

Now, let’s use the str.join() method to add whitespace to that string, which we can do like so:

" ".join(balloon)

If we print this out:

print(" ".join(balloon))

We will see that in the new string that is returned there is added space throughout the first string:

Output:

S a m m y h a s a b a l l o o n

We can also use the str.join() method to return a string that is a reversal from the original string:

print("".join(reversed(balloon)))

Output:

.noollab a sah ymmaS

We did not want to add any part of another string to the first string, so we kept the quotation marks touching with no space in between.

The str.join() method is also useful to combine a list of strings into a new single string.

Let’s create a comma-separated string from a list of strings:

print(",".join(["sharks", "crustaceans", "plankton"]))

Output:

sharks,crustaceans,plankton

If we want to add a comma and a space between string values in our new string, we can simply rewrite our expression with a whitespace after the comma: ", ".join(["sharks", "crustaceans", "plankton"]).

Just as we can join strings together, we can also split strings up. To do this, we will use the str.split() method:

print(balloon.split())

Output:

['Sammy', 'has', 'a', 'balloon.']

The str.split() method returns a list of strings that are separated by whitespace if no other parameter is given.

We can also use str.split() to remove certain parts of an original string.

For example, let’s remove the letter a from the string:

print(balloon.split("a"))

Output:

['S', 'mmy h', 's ', ' b', 'lloon.']

Now the letter a has been removed and the strings have been separated where each instance of the letter a had been, with whitespace retained.

The str.replace() method can take an original string and return an updated string with some replacement.

Let’s say that the balloon that Sammy had is lost. Since Sammy no longer has this balloon, we will change the substring "has" from the original string balloon to "had" in a new string:

print(balloon.replace("has","had"))

Within the parentheses, the first substring is what we want to be replaced, and the second substring is what we are replacing that first substring with. Our output will look like this:

Output:

Sammy had a balloon.

Using the string methods str.join(), str.split(), and str.replace() will provide you with greater control to manipulate strings in Python.

**String constants**

1. **string.ascii\_letters:** Concatenation of the ascii\_lowercase and ascii\_uppercase constants.
2. **string.ascii\_lowercase:** Concatenation of lowercase letters
3. **string.ascii\_uppercase:** Concatenation of uppercase letters
4. **string.digits:** Digit in strings
5. **string.hexdigits:**Hexadigit in strings
6. **string.letters:**concatenation of the strings lowercase and uppercase
7. **string.lowercase:** A string must contain lowercase letters.
8. **string.octdigits:** Octadigit in a string
9. **string.punctuation:** ASCII characters having punctuation characters.
10. **string.printable:** String of characters which are printable
11. **string.endswith:** Returns True if a string ends with the given suffix otherwise returns False
12. **string.isdigit[:](https://www.geeksforgeeks.org/python-string-isdigit-application/)** Returns “True” if all characters in the string are digits, Otherwise, It returns “False”.
13. **string.isalpha[:](https://www.geeksforgeeks.org/python-string-isalpha-application/)** Returns “True” if all characters in the string are alphabets, Otherwise, It returns “False”.
14. **string.index[:](https://www.geeksforgeeks.org/python-string-index-applications/)**Returns the position of the first occurrence of substring in a string
15. **string.uppercase:** A string must contain uppercase letters.
16. **string.whitespace[:](https://www.geeksforgeeks.org/python-string-isspace-application/)** A string containing all characters that are considered whitespace.
17. **string.isdecimal[:](https://www.geeksforgeeks.org/python-string-isdecimal/)**Returns true if all characters in a string are decimal
18. **string.isalnum[:](https://www.geeksforgeeks.org/python-string-isalnum/)**Returns true if all the characters in a given string are alphanumeric.
19. **string.istitle[:](https://www.geeksforgeeks.org/python-string-istitle/)**Returns True if the string is a titlecased string
20. **string.partition[:](https://www.geeksforgeeks.org/string-partition-python/)**splits the string at the first occurrence of the separator and returns a tuple.
21. **string.isidentifier[:](https://www.geeksforgeeks.org/python-string-isidentifier/)**Check whether a string is a valid identifier or not.
22. **string.len[:](https://www.geeksforgeeks.org/python-string-length-len/)**Returns the length of the string.
23. **string.rindex:** Returns the highest index of the substring inside the string if substring is found.
24. **string.max:** Returns the highest alphabetical character in a string.
25. **string.min:** Returns the minimum alphabetical character in a string.
26. **string.splitlines[:](https://www.geeksforgeeks.org/python-string-splitlines/)**Returns a list of lines in the string.
27. **string.capitalize[:](https://www.geeksforgeeks.org/string-capitalize-python/)**Return a word with its first character capitalized.
28. **string.expandtabs:**Expand tabs in a string replacing them by one or more spaces
29. **string.find:**Return the lowest indexin a sub string.
30. **string.rfind[:](https://www.geeksforgeeks.org/python-string-rfind/)**find the highest index.
31. **string.rindex:**Raise ValueError when the substring is not found.
32. **string.count[:](https://www.geeksforgeeks.org/python-string-count/)**Return the number of (non-overlapping) occurrences of substring sub in string
33. **string.lower:**Return a copy of s, but with upper case letters converted to lower case.
34. **string.split[:](https://www.geeksforgeeks.org/python-string-split/)**Return a list of the words of the string,If the optional second argument sep is absent or None
35. **string.rsplit:**Return a list of the words of the string s, scanning s from the end.
36. **string.splitfields:**Return a list of the words of the string when only used with two arguments.
37. **string.join[:](https://www.geeksforgeeks.org/join-function-python/)**Concatenate a list or tuple of words with intervening occurrences of sep.
38. **string.lstrip[:](https://www.geeksforgeeks.org/python-string-strip/)**Return a copy of the string with leading characters removed.
39. **string.rstrip:**Return a copy of the string with trailing characters removed.
40. **string.swapcase[:](https://www.geeksforgeeks.org/python-string-swapcase/)**Converts lower case letters to upper case and vice versa.
41. **string.translate:**translate the characters using table
42. **string.upper:**lower case letters converted to upper case.
43. **string.ljust[:](https://www.geeksforgeeks.org/string-rjust-ljust-python/)**left-justify in a field of given width.
44. **string.rjust[:](https://www.geeksforgeeks.org/string-rjust-ljust-python/)**Right-justify in a field of given width.
45. **string.center:** Center-justify in a field of given width.
46. **string.zfill:** Pad a numeric string on the left with zero digits until the given width is reached.
47. **string.replace[:](https://www.geeksforgeeks.org/replace-in-python-to-replace-a-substring/)**Return a copy of string s with all occurrences of substring old replaced by new.

**Python Lists**

The list is a most versatile data type available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.

Creating a list is as simple as putting different comma-separated values between square brackets.

**Example:**

list1 = ['physics', 'chemistry', 1997, 2000]

list2 = [1, 2, 3, 4, 5 ];

list3 = ["a", "b", "c", "d"]

Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on.

**Accessing Values in Lists:**

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index.

**Example:**

list1 = ['physics', 'chemistry', 1997, 2000]

list2 = [1, 2, 3, 4, 5, 6, 7 ];

print ("list1[0]: ", list1[0])

print ("list2[1:5]: ", list2[1:5])

When the above code is executed, the output is:

list1[0]: physics

list2[1:5]: [2, 3, 4, 5]

**Updating Lists:**

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method.

**Example:**

list = ['physics', 'chemistry', 1997, 2000]

print ("Value available at index 2 : ")

print (list[2])

list[2] = 2001

print ("New value available at index 2 : ")

print (list[2])

When the above code is executed, the output is:

Value available at index 2:

1997

New value available at index 2:

2001

**Delete List Elements:**

To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know.

**Example:**

list1 = ['physics', 'chemistry', 1997, 2000]

print (list1)

del (list1[2])

print ("After deleting value at index 2 : ")

print (list1)

When the above code is executed, the output is:

['physics', 'chemistry', 1997, 2000]

After deleting value at index 2:

['physics', 'chemistry', 2000]

**Basic List Operations:**

Lists respond to the + and \* operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

In fact, lists respond to all of the general sequence operations we used on strings in the prior chapter.

|  |  |  |
| --- | --- | --- |
| **Python Expression** | **Results** | **Description** |
| len([1, 2, 3]) | 3 | Length |
| [1, 2, 3] + [4, 5, 6] | [1, 2, 3, 4, 5, 6] | Concatenation |
| ['Hi!'] \* 4 | ['Hi!', 'Hi!', 'Hi!', 'Hi!'] | Repetition |
| 3 in [1, 2, 3] | True | Membership |
| for x in [1, 2, 3]: print x, | 1 2 3 | Iteration |

**Indexing, Slicing, and Matrixes:**

Because lists are sequences, indexing and slicing work the same way for lists as they do for strings.

Assuming following input −

L = ['spam', 'Spam', 'SPAM!']

|  |  |  |
| --- | --- | --- |
| **Python Expression** | **Results** | **Description** |
| L[2] | 'SPAM!' | Offsets start at zero |
| L[-2] | 'Spam' | Negative: count from the right |
| L[1:] | ['Spam', 'SPAM!'] | Slicing fetches sections |

**Built-in List Functions & Methods:**

Python includes the following list functions:

|  |  |
| --- | --- |
| **S.No.** | **Function with Description** |
| 1 | [**cmp(list1, list2)**](https://www.tutorialspoint.com/python/list_cmp.htm)  Compares elements of both lists. |
| 2 | [**len(list)**](https://www.tutorialspoint.com/python/list_len.htm)  Gives the total length of the list. |
| 3 | [**max(list)**](https://www.tutorialspoint.com/python/list_max.htm)  Returns item from the list with max value. |
| 4 | [**min(list)**](https://www.tutorialspoint.com/python/list_min.htm)  Returns item from the list with min value. |
| 5 | [**list(seq)**](https://www.tutorialspoint.com/python/list_list.htm)  Converts a tuple into list. |

Python includes following list methods

|  |  |
| --- | --- |
| **Sr.No.** | **Methods with Description** |
| 1 | [**list.append(obj)**](https://www.tutorialspoint.com/python/list_append.htm)  Appends object obj to list |
| 2 | [**list.count(obj)**](https://www.tutorialspoint.com/python/list_count.htm)  Returns count of how many times obj occurs in list |
| 3 | [**list.extend(seq)**](https://www.tutorialspoint.com/python/list_extend.htm)  Appends the contents of seq to list |
| 4 | [**list.index(obj)**](https://www.tutorialspoint.com/python/list_index.htm)  Returns the lowest index in list that obj appears |
| 5 | [**list.insert(index, obj)**](https://www.tutorialspoint.com/python/list_insert.htm)  Inserts object obj into list at offset index |
| 6 | [**list.pop(obj=list[-1])**](https://www.tutorialspoint.com/python/list_pop.htm)  Removes and returns last object or obj from list |
| 7 | [**list.remove(obj)**](https://www.tutorialspoint.com/python/list_remove.htm)  Removes object obj from list |
| 8 | [**list.reverse()**](https://www.tutorialspoint.com/python/list_reverse.htm)  Reverses objects of list in place |
| 9 | [**list.sort([func])**](https://www.tutorialspoint.com/python/list_sort.htm)  Sorts objects of list, use compare func if given |

**Python Dictionaries**

Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

**Accessing Values in Dictionary:**

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value.

**Example:**

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

print "dict['Name']: ", dict['Name']

print "dict['Age']: ", dict['Age']

When the above code is executed, the output is:

dict['Name']: Zara

dict['Age']: 7

If we attempt to access a data item with a key, which is not part of the dictionary, we get an error as follows −

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

print "dict['Alice']: ", dict['Alice']

When the above code is executed, the output is:

dict['Alice']:

Traceback (most recent call last):

File "test.py", line 4, in <module>

print "dict['Alice']: ", dict['Alice'];

KeyError: 'Alice'

**Updating Dictionary:**

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown below in the simple example −

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

dict['Age'] = 8; # update existing entry

dict['School'] = "DPS School"; # Add new entry

print "dict['Age']: ", dict['Age']

print "dict['School']: ", dict['School']

When the above code is executed, the output is:

dict['Age']: 8

dict['School']: DPS School

**Delete Dictionary Elements**

You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation.

To explicitly remove an entire dictionary, just use the del statement.

**Example:**

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

del dict['Name']; # remove entry with key 'Name'

dict.clear(); # remove all entries in dict

del dict ; # delete entire dictionary

print "dict['Age']: ", dict['Age']

print "dict['School']: ", dict['School']

This produces the following result. Note that an exception is raised because after del dict dictionary does not exist any more:

dict['Age']:

Traceback (most recent call last):

File "test.py", line 8, in <module>

print "dict['Age']: ", dict['Age'];

TypeError: 'type' object is unsubscriptable

**Note** − del() method is discussed in subsequent section.

**Properties of Dictionary Keys:**

Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects. However, same is not true for the keys.

There are two important points to remember about dictionary keys:

1. More than one entry per key not allowed. Which means no duplicate key is allowed. When duplicate keys encountered during assignment, the last assignment wins.

Example:

dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'}

print "dict['Name']: ", dict['Name']

When the above code is executed, the output is:

dict['Name']: Manni

1. Keys must be immutable. Which means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed.

Example:

dict = {['Name']: 'Zara', 'Age': 7}

print "dict['Name']: ", dict['Name']

When the above code is executed, the output is:

Traceback (most recent call last):

File "test.py", line 3, in <module>

dict = {['Name']: 'Zara', 'Age': 7};

TypeError: list objects are unhashable

**Built-in Dictionary Functions & Methods:**

|  |  |
| --- | --- |
| **S.No.** | **Function with Description** |
| 1 | **cmp(dict1,dict2)**  Compares elements of both dict. |
| 2 | **len(dict)**  Gives the total length of the dictionary. This would be equal to the number of items in the dictionary. |
| 3 | **str(dict)**  Produces a printable string representation of a dictionary |
| 4 | **type(variable)**  Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type. |

Python includes the following dictionary functions −

Python includes following dictionary methods −

|  |  |
| --- | --- |
| **S.No.** | **Methods with Description** |
| 1 | dict.clear()  Removes all elements of dictionary dict |
| 2 | dict.copy()  Returns a shallow copy of dictionary dict |
| 3 | dict.fromkeys()  Create a new dictionary with keys from seq and values set to value. |
| 4 | dict.get(key, default=None)  For key key, returns value or default if key not in dictionary |
| 5 | dict.has\_key(key)  Returns true if key in dictionary dict, false otherwise |
| 6 | dict.items()  Returns a list of dict's (key, value) tuple pairs |
| 7 | dict.keys()  Returns list of dictionary dict's keys |
| 8 | dict.setdefault(key, default=None)  Similar to get(), but will set dict[key]=default if key is not already in dict |
| 9 | dict.update(dict2)  Adds dictionary dict2's key-values pairs to dict |
| 10 | dict.values()  Returns list of dictionary dict's values |